Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel

Jia-Ju Bai, Julia Lawall, Shi-Min Hu

Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel*

JIA-JU BAI, Tsinghua University, China
JULIA LAWALL, Sorbonne University/Inria/LIP6, France
SHI-MIN HU, Tsinghua University, China

Atomic context is an execution state of the Linux kernel, in which kernel code monopolizes a CPU core. In this state, the Linux kernel may only perform operations that cannot sleep, as otherwise a system hang or crash may occur. We refer to this kind of concurrency bug as a sleep-in-atomic-context (SAC) bug. In practice, SAC bugs are hard to find, as they do not cause problems in all executions.

In this paper, we propose a practical static approach named DSAC, to effectively detect SAC bugs in the Linux kernel. DSAC uses three key techniques: (1) a summary-based analysis to identify the code that may be executed in atomic context, (2) a connection-based alias analysis to identify the set of functions referenced by a function pointer, and (3) a path-check method to filter out repeated reports and false bugs. We evaluate DSAC on Linux 4.17, and find 1159 SAC bugs. We manually check all the bugs, and find that 1068 bugs are real. We have randomly selected 300 of the real bugs and sent them to kernel developers. 220 of these bugs have been confirmed, and 51 of our patches fixing 115 bugs have been applied.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; *Operating systems*; • **Computer systems organization** → Reliability;

Additional Key Words and Phrases: Static analysis, bug detection, atomic context, operating system

ACM Reference Format:
Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2020. Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel. *ACM Trans. Comput. Syst.* 36, 4, Article 10 (April 2020), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Concurrency bugs are known to be difficult to detect in the Linux kernel, because most of them are hard to reproduce in real execution. Many tools have been proposed to detect common concurrency bugs, such as atomicity violations and data races. However, as another kind of concurrency bug, sleep-in-atomic-context (SAC) bugs have received less attention. SAC bugs occur at the kernel level when a sleeping operation is performed in atomic context [18], such as when holding a spinlock or executing an interrupt handler. Code executing in atomic context monopolizes a CPU core, and the progress of other threads that need to concurrently access the same resources is delayed. Thus, the code executing in atomic context should complete as quickly as possible. Sleeping in atomic context is forbidden, because it can block a CPU core for a long time and may lead to a system hang.

*This paper extends a previous conference paper [7] in the 2018 USENIX Annual Technical Conference. This paper includes a new summary-based analysis to identify atomic context, a new connection-based alias analysis to handle function-pointer calls, and an improved path-check method to filter out repeated reports and false bugs. Compared to that work, this paper analyzes the whole kernel instead of only kernel modules, and handles function-pointer calls. This paper also finds more real bugs in a more recent Linux kernel version.

Authors' addresses: Jia-Ju Bai, Tsinghua University, Beijing, 100084, China, baijiaju1990@gmail.com; Julia Lawall, Sorbonne University/Inria/LIP6, Paris, France, julia.lawall@lip6.fr; Shi-Min Hu, Tsinghua University, Beijing, 100084, China, shimin@tsinghua.edu.cn.

Even though sleeping in atomic context is forbidden, many SAC bugs still exist in the Linux kernel. The main reasons why SAC bugs continue to occur include: (1) Determining whether an operation can sleep often requires system-specific experience; (2) Some parts of the kernel code can be difficult to test, for example, testing a device driver requires its associated device; (3) SAC bugs do not always cause problems in real execution, and they are often hard to reproduce at runtime. Recent studies [19, 60] have shown that SAC bugs have caused serious system hangs at runtime. Thus, it is necessary to design approaches to detect them.

Many existing approaches [13, 26, 37, 55] can detect concurrency bugs, but most of them are designed for user-level applications. Some approaches [20, 25, 27, 54, 57] can detect some common kinds of kernel-level concurrency bugs, such as atomicity violations and data races, but they have not addressed SAC bugs. Several approaches [4, 15, 24, 46] can detect common kinds of OS kernel faults, including SAC bugs. But they are not specific to SAC bugs, and most of them [15, 24, 46] are designed to collect statistics rather than report specific bugs to the user, making issues such as detection time and false positive rate less important. Besides, these approaches have not considered function pointers, so they may miss SAC bugs that involve function-pointer calls.

In this paper, we propose a static approach named DSAC[1] that targets accurately and efficiently detecting SAC bugs in the Linux kernel. Specifically, DSAC consists of four phases. Firstly, DSAC compiles the source code of the Linux kernel and records the link information. Secondly, DSAC traverses the Linux kernel code to collect some useful information, such as the locations, callers and callees of direct function calls. Thirdly, using the collected information, for each source file, DSAC identifies the source files to which it is connected according to link information and direct function calls. This *connection information* is used to guide function-pointer analysis in the next phase. Fourthly, DSAC uses a *summary-based analysis* to identify the code that may be executed in atomic context. To accurately cover as much code as possible, this analysis is inter-procedural, context-sensitive and flow-sensitive. To handle function-pointer calls and find deep SAC bugs, DSAC uses a *connection-based alias analysis* to identify the set of functions referenced by the function pointer. In this phase, DSAC also uses a *path-check method* to filter out repeated reports and false bugs. We have implemented DSAC using LLVM [39]. DSAC works automatically with the given Linux kernel source code. Moreover, it can work in parallel to speed up the analysis.

Overall, we make the following contributions:

- We first analyze the main challenges of detecting SAC bugs in the Linux kernel, and then propose three key techniques to address these challenges: (1) a summary-based analysis to identify the code that may be executed in atomic context; (2) a connection-based alias analysis to identify the set of functions referenced by a function pointer; (3) a path-check method to filter out repeated reports and false bugs.
- Based on the three techniques, we propose a practical approach named DSAC, to effectively and automatically detect SAC bugs in the Linux kernel.
- We evaluate DSAC on a Linux stable version 3.17.2 and a mainline release 4.17, and find 891 and 1159 bugs respectively. We manually check these bugs, and find that 805 and 1068 are real, respectively, in these versions, giving false positive rates of 9.7% and 7.8%. 304 bugs real bugs in Linux 3.17.2 have been fixed in 4.17. We have randomly selected 300 of the real bugs in Linux 4.17, and sent them to kernel developers. 220 of these bugs have been confirmed, and 51 of our patches fixing 115 bugs have been applied.

The remainder of this paper is organized as follows. Section 2 presents the background. Section 3 introduces the main challenges in detecting SAC bugs. Section 4 introduces our key techniques to address these challenges. Section 5 introduces DSAC in detail. Section 6 presents the evaluation.

[1]DSAC website: https://oslab.cs.tsinghua.edu.cn/DSAC2/index.html

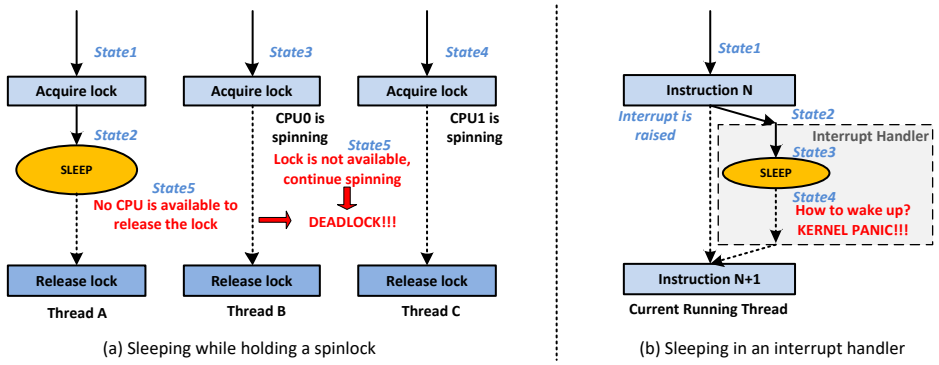


Fig. 1. Side effects caused by sleeping in atomic context.

Section 7 compares DSAC to previous approaches. Section 8 discusses some possible extensions for DSAC. Section 9 gives the related work and Section 10 concludes the paper.

2 BACKGROUND

In this section, we first introduce atomic context, and then motivate our work by a real SAC bug in the Linux kernel.

2.1 Atomic Context

Atomic context is an OS kernel state in which a CPU core is monopolized to execute code, and the progress of other threads that need to concurrently access the same resources is delayed. This context is used to protect resources from concurrent access. In the Linux kernel, there are two common examples of atomic context, namely *holding a spinlock* and *executing an interrupt handler*. In atomic context, code execution should complete as quickly as possible without being able to be rescheduled. Due to this special situation, *sleeping in atomic context is not allowed*, because it can block CPU cores for long periods and may lead to a system hang or crash.

In Figure 1, we use the two common instances of atomic context to explain the side effects of sleeping in atomic context:

Sleeping while holding a spinlock. Suppose that a thread A sleeps while holding a spinlock. If another thread B requests the same spinlock at that time, thread B will monopolize a CPU to spin until the spinlock is released by thread A. If all CPUs are monopolized to spin like thread B, there will be no CPU on which to wake up thread A to release the spinlock, and thus a deadlock will occur, causing a system hang. In the single-core system, sleeping while holding a spinlock should deterministically cause a deadlock at runtime, because only one CPU is available, and it has been monopolized by thread B. In a multi-core system, sleeping while holding a spinlock does not always cause a deadlock at runtime, because another CPU may be available for thread A on which it can release the spinlock. Figure 1(a) shows an example of the deadlock caused by sleeping while holding a spinlock. The system has two CPUs and runs from `State1` to `State5`. The deadlock finally occurs in `State5`.

Sleeping in an interrupt handler. To quickly respond to interrupts, when an interrupt is raised, the Linux kernel executes the interrupt handler in the context of the currently running thread. Thus, the interrupt handler is not associated with a fixed process or thread context. For this reason, if the interrupt handler sleeps during execution, the OS scheduler cannot reschedule it because it does not have a backing process or thread, implying that it is not a schedulable entity. In this situation, a kernel panic will occur, causing a system crash. Figure 1(b) shows how a kernel panic is

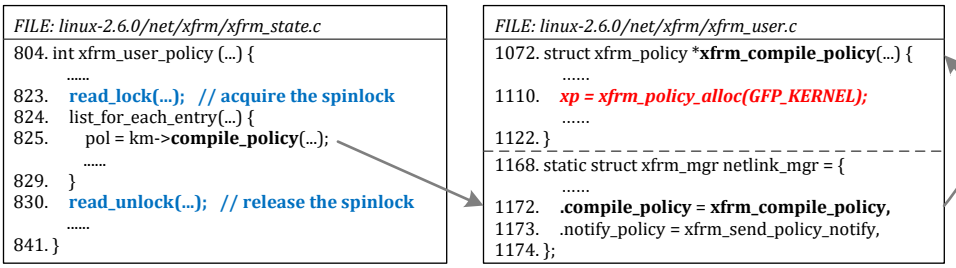


Fig. 2. Example SAC bug in the *xfrm* network module of Linux 2.6.0.

caused by sleeping in an interrupt handler. The system runs from `State1` to `State4`, and a kernel panic finally occurs in `State4`.

Note that atomic context only occurs at the kernel level, because user-level applications are regularly interrupted by the OS scheduler when their time slice ends. Though kernel developers often know that sleeping is not allowed in atomic context, many SAC bugs still exist in the Linux kernel [24, 46].

2.2 Motivating Example

Figure 2 shows a reported SAC bug in the Linux *xfrm* network module. This bug was introduced when the module was integrated in Linux 2.6.0 (released in December 2003), and was fixed in Linux 2.6.36 (released in October 2010), nearly 7 years later. The function `xfrm_user_policy` calls `read_lock` to acquire a spinlock on line 823, and then calls a function pointer `km->compile_policy` on line 825. This function pointer is assigned to the function `xfrm_compile_policy` in another source file. The function `xfrm_compile_policy` calls `xfrm_policy_alloc` with `GFP_KERNEL` to allocate a policy-related data structure on line 1110. According to the Linux kernel documentation [1], `xfrm_policy_alloc` called with `GFP_KERNEL` can sleep at runtime, thus this code contains a SAC bug. To fix this bug, the commit `2f09a4d5daaa`[2] replaced `GFP_KERNEL` with `GFP_ATOMIC`, which prevents the allocation from sleeping.

This example illustrates four main reasons why SAC bugs occur in the Linux kernel. (1) Determining whether an operation can sleep requires OS-specific knowledge. In this example, without experience in Linux kernel development, it may be hard to know that the function `xfrm_policy_alloc` called with `GFP_KERNEL` can sleep at runtime. (2) SAC bugs do not always cause problems in real execution and are hard to reproduce at runtime. In this example, the function `xfrm_policy_alloc` called with `GFP_KERNEL` only sleeps when memory is insufficient. Even in a low-memory situation, this SAC bug is not always triggered at runtime in a multi-core system, because of the non-determinism of concurrent execution. (3) Inter-procedural properties and function pointers need to be considered when finding SAC bugs. In this example, the function `xfrm_policy_alloc` is called through `xfrm_compile_policy` referenced by a function pointer, after `read_lock` is called. In fact, a main reason why this bug persisted for a long time may be that it involves a function-pointer call.

Some recent studies [19, 60] have shown that SAC bugs have caused serious system hangs, and these bugs are often hard to locate and reproduce. Thus, to improve the reliability of the Linux kernel, it is necessary to design an effective approach to detect SAC bugs.

3 CHALLENGES

There are three main challenges in detecting SAC bugs in the Linux kernel.

[2]Patch link: http://lists.openwall.net/netdev/2010/08/13/47

3.1 C1: Accuracy and Efficiency in Code Analysis

When identifying the code that may be executed in atomic context, the code analysis should be accurate and efficient. Flow-sensitive static analysis searches each code path with the goal of providing accuracy, but it is often inefficient in analyzing large software. The Linux kernel code base is very large (amounting to over 16M lines of source code, measured with CLOC [17], in our tested version Linux 4.17) and complex, and it contains many source files in multiple directories. Thus, directly using flow-sensitive analysis to check the whole Linux kernel may be quite time consuming.

3.2 C2: Handling Function Pointers

The Linux kernel is essentially an object-oriented system, in which data structures containing function pointers play the role of objects containing methods. This design improves extensibility, but greatly complicates static analysis. Indeed, many possible execution paths contain calls to function pointers, and static analysis needs to correctly determine the set of functions that may be referenced by these pointers. An accurate function pointer analysis is thus very important when detecting SAC bugs. On the one hand, identifying too few or no referenced functions implies that SAC bugs like the one shown in Figure 2 that hide in the missed functions cannot be detected. On the other hand, if incorrect functions are identified for function pointers, false SAC bugs may be reported.

3.3 C3: Filtering out Repeated and False Bugs

Flow-sensitive analysis may detect repeated bugs when these bugs take the spinlock at the same place and call the same sleep-able function but only differ in their code paths. Moreover, flow-sensitive analysis may detect false bugs, because the analysis does not consider variable values when analyzing path conditions, and thus may search some infeasible code paths.

4 KEY TECHNIQUES

To solve the above challenges, we propose three key techniques. For *C1*, we propose a summary-based flow-sensitive analysis to accurately and efficiently identify the code that may be executed in atomic context. For *C2*, we propose a connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. For *C3*, we propose a path-check method to filter out repeated reports and false bugs. We now introduce these techniques.

4.1 Summary-Based Analysis

Our summary-based analysis identifies the code in atomic context, and it has the following properties: (1) The analysis is context-sensitive and inter-procedural, in order to maintain the spinlock status and detect atomic context across functions calls. (2) The analysis is flow-sensitive, in order to improve the accuracy. (3) The analysis uses function summaries to reduce repeated analysis, which can improve the efficiency. (4) The analysis uses a connection-based alias analysis (which will be introduced in Section 4.2) to handle function-pointer calls. (5) When the analysis encounters a call to a function that can sleep at runtime (a sleep-able function) in atomic context, the analysis uses a path-check method (which will be introduced in Section 4.3) to filter out repeated reports and false bugs.

Our summary-based analysis has two steps. The first step identifies interrupt handler functions and calls to spin-lock functions, which initiate atomic context. For interrupt handler functions, we identify the calls to interrupt-handler-register kernel interfaces (like `request_irq` and

```
HandleCall(mycall, path_stack, lock_stack)
 1: if lock_stack == ø and g_intr_flag == FALSE then
 2:     return;
 3: end if
 4: func_set := ø;
 5: if mycall is a call to a function pointer then
 6:     /* Use connection-based alias analysis to
 7:      * get the set of referenced functions */
 8:     func_set := ConnAliasAnalysis(mycall, g_cur_file);
 9: else
10:     myfunc := GetCalledFunction(mycall);
11:     Store myfunc in func_set;
12: end if
13: foreach func in func_set do
14:     HandleFunc(func, path_stack, lock_stack);
15: end foreach
```

```
HandleFunc(myfunc, path_stack, lock_stack)
 1: /* Search existing function summaries */
 2: if FindFuncSummary(myfunc, lock_stack, g_intr_flag) == TRUE then
 3:     /* Get existing atomic context path in function summary */
 4:     atomic_path_stack_set = GetAtomicPathInSummary(myfunc);
 5:     if atomic_path_stack_set == ø then
 6:         return;
 7:     end if
 8:     foreach atomic_path_stack in atomic_path_stack_set do
 9:         /* Splice and get full atomic context path */
10:         my_path_stack := path_stack + atomic_path_stack;
11:         last_call := GetLastFuncCall(my_path_stack);
12:         /* Use path-check method to validate path feasibility */
13:         if PathCheck(last_call, my_path_stack) == TRUE then
14:             WriteBugFile(last_call, my_path_stack);
15:         end if
16:     end foreach
17: else
18:     /* Add a new function summary */
19:     CreateFuncSummary(myfunc, lock_stack, g_intr_flag);
20:     entry_block := GetEntryBlock(myfunc);
21:     HandleBlock(entry_block, path_stack, lock_stack);
22: end if
```

```
HandleBlock(myblock, path_stack, lock_stack)
 1: if PathHasExisted(myblock, path_stack, lock_stack) == TRUE then
 2:     return; /* Prevent infinite handling on loops and recursive calls */
 3: end if
 4: AddPathStack(myblock, path_stack);
 5: foreach func_call in FunctionCallList(myblock) do
 6:     if func_call is a call to a spin-lock function then
 7:         Push func_call onto lock_stack;
 8:     else if func_call is a call to a spin-unlock function then
 9:         Pop an item from lock_stack;
10:     else if func_call is a call to a basic sleep-able function or
11:             a function with a sleep-able constant flag then
12:         /* Use path-check method to validate path feasibility */
13:         if PathCheck(func_call, path_stack) == TRUE then
14:             WriteBugFile(func_call, path_stack);
15:         end if
16:         /* Store atomic context path in function summary */
17:         myfunc := GetParentFunc(myblock);
18:         StoreAtomicPathInSummary(func_call, path_stack, myfunc);
19:     else
20:         HandleCall(func_call, path_stack, lock_stack);
21:     end if
22: end foreach
23: if lock_stack == ø and g_intr_flag == FALSE then
24:     return;
25: end if
26: foreach block in SuccessorBlocks(myblock) do
27:     HandleBlock(block, path_stack, lock_stack);
28: end foreach
```

```
CodeAnalysis: Identify the code in atomic context
 1: InitFuncSummary();
 2: g_cur_file := GetCurrentSourceFileName();
 3: foreach func_call in spinlock_func_set do
 4:     lock_stack := ø; path_stack := ø; g_intr_flag := FALSE;
 5:     myblock := GetBasicBlock(func_call);
 6:     HandleBlock(block, path_stack, lock_stack);
 7: end foreach
 8: foreach func in intr_handler_func_set do
 9:     lock_stack := ø; path_stack := ø; g_intr_flag := TRUE;
10:     entry_block := GetEntryBlock(func);
11:     HandleBlock(entry_block, path_stack, lock_stack);
12: end foreach
```

Fig. 3. Procedure of summary-based analysis.

`tasklet_init` in the Linux kernel), and extract interrupt handler functions from the relevant arguments. For calls to spin-lock functions, we identify them according to the function name.

The second step performs the main analysis, *CodeAnalysis*, which is defined in Figure 3. The analysis maintains two stacks, namely a path stack (*path_stack*) to store the executed code path (a sequence of locations of basic blocks and function calls) and a lock stack (*lock_stack*) to store the spinlock status. For simplicity, the analysis only considers the number and locations of locks in *lock_stack*, instead of the alias of each lock. The analysis also uses a global flag (*g_intr_flag*) to indicate whether the code is in an interrupt handler. If *lock_stack* is not empty or *g_intr_flag* is *TRUE*, the code may be executed in atomic context. Besides, the analysis creates and stores a function summary for each analyzed function for each execution context. This function summary contains the location of the function, the *lock_stack* and *g_intr_flag* for which the function is analyzed, and code paths of the SAC bugs found in the function (we refer to such a code path as an *atomic-context code path*). *CodeAnalysis* uses *HandleCall* to handle a function call, *HandleBlock* to handle a basic block, and *HandleFunc* to handle the definition of a called function.

HandleCall. It handles the function call *mycall* with the arguments *path_stack* and *lock_stack*. Firstly, *HandleCall* checks whether *lock_stack* is empty and *g_intr_flag* is *FALSE* (lines 1-3). If so, no spinlock is held and the code is not in an interrupt handler, and thus *HandleCall* returns. Secondly, *HandleCall* identifies the set of functions called by *mycall* (lines 5-12), which is represented by *func_set*. If *mycall* is a function-pointer call, *HandleCall* performs a connection-based alias analysis, described in Section 4.2, to identify the set of functions referenced by the function pointer, and

stores them in *func_set*. Otherwise, *HandleCall* only stores the called function *myfunc* in *func_set*. Finally, *HandleCall* handles each function in *func_set* by using *HandleFunc* (lines 13-15).

HandleBlock. It handles the basic block *myblock* with the arguments *path_stack* and *lock_stack*. Firstly, *HandleBlock* searches *path_stack* to check whether *myblock* has been analyzed (lines 1-3) with respect to the current *lock_stack*. If so, it returns to avoid repeated analysis. Secondly, *HandleBlock* adds *myblock* into *path_stack* (line 4). Thirdly, *HandleBlock* handles each function call *func_call* in *myblock* (lines 5-22) in order. If *func_call* is a call to a spin-lock or spin-unlock function, *HandleBlock* respectively pushes the call onto or pops an item from *lock_stack*. If *func_call* is a call to a basic sleep-able function or a function with a sleep-able constant flag, *HandleBlock* uses a path-check method, described in Section 4.3, to validate whether the code path is feasible. If so, a SAC bug is found, and its call path is written into the bug report file. Then, *HandleBlock* stores the code path *path_stack* reaching *func_call* as an atomic-context code path in the summary of *myblock*'s function *myfunc*, and this atomic-context code path is used to build the completed code path in the summary-based analysis. If *func_call* is not a call to a spin-lock, spin-unlock, basic sleep-able function or a function with a sleep-able constant flag, *HandleBlock* uses *HandleCall* to handle *func_call*. Fourthly, *HandleBlock* checks whether *lock_stack* is empty and *g_intr_flag* is *FALSE* (lines 23-25). If so, it returns. Finally, *HandleBlock* handles each successor basic block of *myblock* by using *HandleBlock*.

HandleFunc. It handles the function *myfunc* with the arguments *path_stack* and *lock_stack*. *HandleFunc* searches existing function summaries stored in the database, to check whether *myfunc* has been already handled under the current execution context (line 1). If so, *HandleFunc* uses the results produced by the previous analysis (lines 3-16). *HandleFunc* searches the function summary of *myfunc* to find atomic-context code paths starting from *myfunc*, and stores the code paths in a set *atomic_path_stack_set*. If no code path is found, namely *myfunc* contains no calls to sleep-able functions, then *HandleFunc* returns directly. Then, *HandleFunc* handles each code path *atomic_path_stack* in the set *atomic_path_stack_set*. It splices *path_stack* and *atomic_path_stack* to build a complete atomic context code path *my_path_stack*. Later, *HandleFunc* gets the last function call from *my_path_stack* as the end of the code path, and uses the path-check method, described in Section 4.3, to validate whether the code path is feasible. If so, a SAC bug is found, and its call path is written into the bug report file. If no function summary is found, it indicates that *myfunc* has not been handled under the current execution context. In this case, *HandleFunc* creates a new function summary for *myfunc* for the current execution context, stores the summary in the database, and analyzes the definition of *myfunc* starting from its entry basic block by using *HandleBlock*.

CodeAnalysis. It performs the main analysis, in three steps. Firstly, *CodeAnalysis* initializes the set of function summaries. Secondly, *CodeAnalysis* handles each call to a spin-lock function in the kernel code. It clears *lock_stack* and *path_stack*, and sets *g_intr_flag* to *FALSE*, and then starts the analysis by using *HandleBlock* to handle the basic block containing the call. Thirdly, *CodeAnalysis* handles each interrupt handler function. It clears *lock_stack* and *path_stack*, and sets *g_intr_flag* to *TRUE*, and then starts the analysis by using *HandleBlock* to handle the entry basic block of the function.

Our summary-based analysis has four main advantages: (1) By using flow-sensitive and context-sensitive analysis, our analysis can accurately identify the code that may be executed in atomic context. (2) By using function summaries and storing atomic-context code paths, our analysis can effectively reduce repeated analysis and utilize previous analysis results, to improve efficiency without damaging accuracy. (3) Our analysis is inter-procedural and can analyze functions across different source files and modules. (4) Our analysis handles function-pointer calls, which are not considered in existing approaches that detect SAC bugs [15, 24, 46]. However, a main limitation of

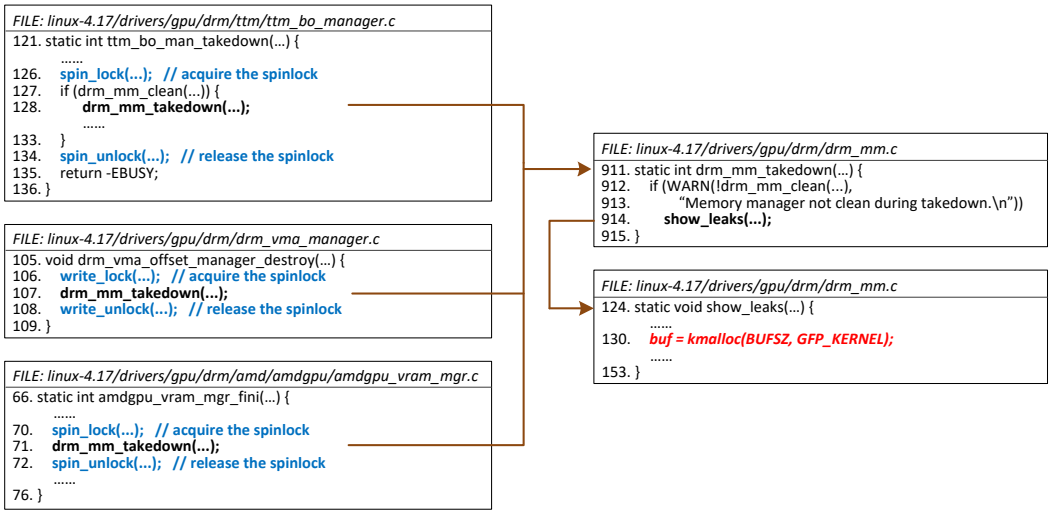
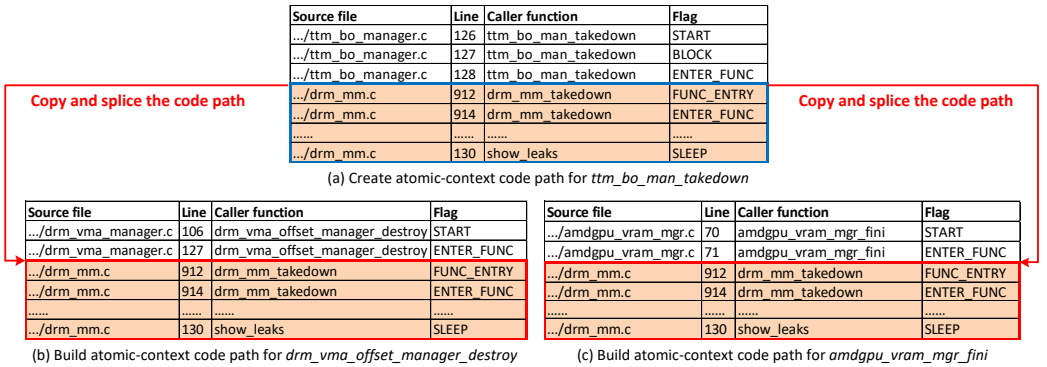


Fig. 4. Examples of summary-based analysis.



Source file	Line	Caller function	Flag
.../ttm_bo_manager.c	126	ttm_bo_man_takedown	START
.../ttm_bo_manager.c	127	ttm_bo_man_takedown	BLOCK
.../ttm_bo_manager.c	128	ttm_bo_man_takedown	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
.....	.....	.....	.....
.../drm_mm.c	130	show_leaks	SLEEP

(a) Create atomic-context code path for *ttm_bo_man_takedown*

Source file	Line	Caller function	Flag
.../drm_vma_manager.c	106	drm_vma_offset_manager_destroy	START
.../drm_vma_manager.c	127	drm_vma_offset_manager_destroy	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
.....	.....	.....	.....
.../drm_mm.c	130	show_leaks	SLEEP

(b) Build atomic-context code path for *drm_vma_offset_manager_destroy*

Source file	Line	Caller function	Flag
.../amdgpu_vram_mgr.c	70	amdgpu_vram_mgr_fini	START
.../amdgpu_vram_mgr.c	71	amdgpu_vram_mgr_fini	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
.....	.....	.....	.....
.../drm_mm.c	130	show_leaks	SLEEP

(c) Build atomic-context code path for *amdgpu_vram_mgr_fini*

Fig. 5. Examples of atomic-context code paths.

our analysis is that variable value information is not considered, which may cause false positives in bug detection.

To illustrate our summary-based analysis, we consider the Linux kernel code shown in Figure 4. The functions `ttm_bo_man_takedown`, `drm_vma_offset_manager_destroy` and `amdgpu_vram_mgr_fini` all call the function `drm_mm_takedown` while holding a spinlock. The function `drm_mm_takedown` calls `show_leaks`, and the function `show_leaks` calls `kmalloc` with `GFP_KERNEL`, which can sleep at runtime resulting in three SAC bugs. In this example, the function `drm_mm_takedown` is called under the same lock status, namely holding one spinlock, thus this function can be just analyzed once, instead of three times.

For the example in Figure 4, assume that our summary-based analysis first handles the function `ttm_bo_man_takedown`, and creates the function summary for `drm_mm_takedown` while holding one spinlock. Our analysis stores the atomic-context code path in the function summary for `drm_mm_takedown` that is shown in Figure 5(a). Each item in the code path represents the code location of a basic block or function call that has to be traversed to reach the current point in the code. The item consists of four fields, namely the source file name, source line number, caller function

and a flag. The flag indicates the analysis state of the code location. For example, in Figure 5, "START" means the location of the start of the analysis; "BLOCK" means the entry of a basic block; "ENTER_FUNC" means the location of a function call whose function definition is to be analyzed; "FUNC_ENTRY" means the entry of a function; ""SLEEP" means the location of a sleepable function call. The called function and the flag are not strictly necessary for the algorithm, but facilitate generating an understandable report when a bug is found. When our analysis handles the functions `drm_vma_offset_manager_destroy` and `amdgpu_vram_mgr_fini`, it finds that the function `drm_mm_takedown` with holding one spinlock has been already analyzed, by searching the function summaries. Then, our analysis finds the atomic-context code paths stored in the summary of `drm_mm_takedown`. Later, as shown in Figure 5(b) and 5(c), our analysis copies this code path and splices it with the handled code paths in the functions `drm_vma_offset_manager_destroy` and `amdgpu_vram_mgr_fini`, to build complete atomic-context code paths. Finally, our analysis uses a path check method to perform code path validation, and finds that the code path is feasible. Thus, our analysis writes the bug information and code path into the bug report file.

4.2 Connection-Based Function-Pointer Analysis

Overall, we identify the set of functions called by a function-pointer call through two steps. Firstly, before the summary-based analysis, we collect the functions that are assigned to function pointers. Secondly, during the summary-based analysis, we select the correct function(s) at each given call site from the collected functions. The first step is performed straightforwardly, by scanning the assignments to function pointers in the Linux code. The second step, of how to identify the correct function(s) at a particular call site, is more difficult.

To illustrate the problem with the second step, we consider the Linux kernel code shown in Figure 6. Figure 6(a) shows that in the *e1000* driver, the function `e1000_dump_eeprom` calls a function pointer through the `get_eeprom` field of a `struct ethtool_ops` data structure (line 748). This field is initialized in multiple drivers to different functions. Figure 6(b) shows three candidate functions: `e1000_get_eeprom`, `jme_get_eeprom` and `sky2_get_eeprom`. Because this function pointer is called in the *e1000* driver, the referenced function(s) should be related to this driver. From the Makefile of the e1000 drivers shown in Figure 6(c), *e1000_main.c* is linked with *e1000_ethtool.c* in the same kernel module, thus `e1000_get_eeprom` should be the correct function referenced by the function pointer.

From this example, we observe that *link information* can be used to identify the correct function(s) referenced by a function-pointer call. The simplest case is that the candidate function and the function-pointer call are in the same source file. In other cases, *for a candidate function F that may be referenced by a function pointer, if its source file is linked with the source file of a function-pointer call, F is very likely to be a correct referenced function for the function-pointer call.* The link information reflects the *connection* between the source files of the candidate function and function-pointer call. This connection is strong, as the involved two source files are linked in the same kernel module.

Besides calling function pointers in the same kernel module, in the Linux kernel, one kernel module can also call the functions in another kernel module through function pointers. In this case, the link information is not sufficient to identify the correct function(s) referenced by a function-pointer call, because the related kernel modules may not be explicitly linked together. To handle this case, we observe that direct function calls can be used to identify the connection between the source files of the candidate function and function-pointer call. Specifically, *if there exists a direct function call between File1 and File2 (File1 → File2 or File2 → File1), then we say that File1 and File2 have a direct connection.* This connection can be transitive. Namely, if there exists a direct function call from *File1* to *File2* (*File1* → *File2*) and from *File2* to *File3* (*File2* → *File3*), then we say that *File1* and *File3* have an indirect connection (*File1* → *File3*). This connection of function call is weaker than

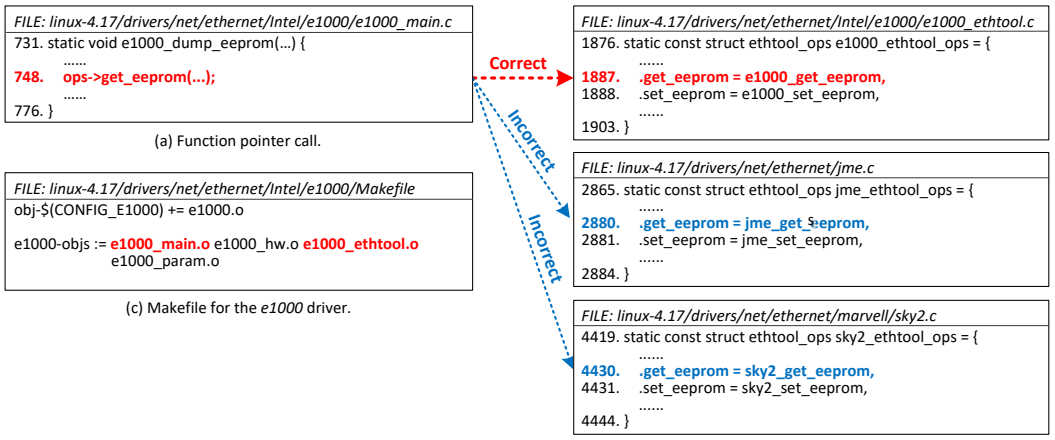


Fig. 6. Examples of handling function-pointer calls through link information.

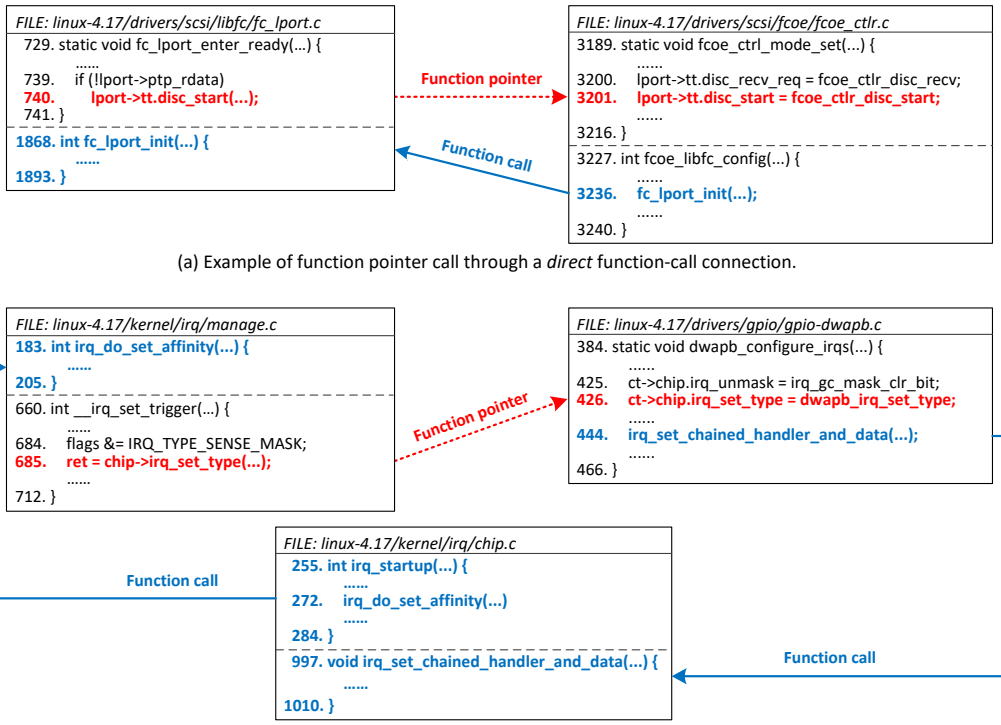


Fig. 7. Examples of handling function-pointer calls through function-call connections.

the link-information connection, as the involved two source files are in different kernel modules. For this reason, the link-information connection is preferentially used for function-pointer analysis.

To illustrate the function-call connection between different kernel modules, we consider two examples in the Linux kernel code shown in Figure 7.

Figure 7(a) shows an example of a direct function-call connection. The driver *libfc* uses a function-pointer call on line 740 in *fc_lport.c*, and the driver *fcoe* has an assignment of this function pointer to `fcoe_ctlr_disc_start` on line 3201 in *fcoe_ctlr.c*. In *fcoe_ctlr.c*, the function `fcoe_libfc_config` calls the function `fc_lport_init` defined in *fc_lport.c*, thus the two source files have a direct connection (*fcoe_ctlr.c* → *fc_lport.c*). For this reason, `fcoe_ctlr_disc_start` in *fcoe_ctlr.c* should be a function referenced by the function-pointer call in *fc_lport.c*.

Figure 7(b) shows an example of an indirect function-call connection. The kernel uses a function-pointer call on line 685 in *manage.c*, and the driver *gpio* has an assignment of this function pointer to `dwapb_irq_set_type` on line 426 in *gpio-dwapb.c*. In *gpio-dwapb.c*, there is no function that calls a function defined in *manage.c*, so the two source files do not have a direct function-call connection. However, the function `dwapb_configure_irqs` calls the function `irq_set_chained_handler_and_data` defined in *chip.c* (*gpio-dwapb.c* → *chip.c*), and in *chip.c*, the function `irq_startup` calls the function `irq_do_set_affinity` defined in *manage.c* (*chip.c* → *manage.c*), so *gpio-dwapb.c* and *manage.c* have an indirect function-call connection (*gpio-dwapb.c* → *chip.c*). For this reason, `dwapb_irq_set_type` in *gpio-dwapb.c* should be a function referenced by the function-pointer call in *manage.c*.

Based on the connections of link information and function calls, we propose a connection-based alias analysis to identify the set of functions referenced by a function pointer. Specifically, our analysis has two steps:

The first step is to collect candidate functions referenced by function pointers and the connections between source files. This step is performed before the summary-based analysis described in Section 4.1. To collect candidate functions, our analysis traverses the Linux kernel code and handles function-pointer assignments. Specifically, our analysis focuses on function-pointer assignments involving data-structure fields, for two reasons. Firstly, many function pointers in kernel code are obtained from data-structure fields [9], especially in device drivers, as illustrated in Figure 6 and Figure 7. We have studied the function-pointer calls in the x86 code of Linux 4.17, and find that over 95% (*i.e.*, 34,452 out of 36,174) involve functions stored in data-structure fields. Secondly, a function pointer stored in a data-structure field can be explicitly can be distinguished from other function pointers with the data-structure type, field offset and function-pointer type. To collect the connections between source files, our analysis records the link information during kernel code compilation and records direct function calls between different source files. All of this information is stored in the database.

The second step is to identify the set of functions referenced by a function pointer, which is done during the summary-based analysis described in Section 4.1. Figure 8 shows the main procedure *FuncPtrAnalysis*. The inputs are a function-pointer call *func_ptr_call* and the name of its source file *src_file* where the summary-based analysis starts. The output is *func_set*, the set of functions referenced by the function pointer for *func_ptr_call*. Firstly, our analysis clears *func_set*, and gets the called function pointer *func_ptr* of *func_ptr_call* (lines 1-2). Secondly, by looking up the collected information stored in the database, our analysis gets the set of candidate functions for *func_ptr*, which is represented as *cand_func_set* (line 3). Because our analysis focuses on function pointers stored in data-structure fields, we perform field-based analysis [30] here. Specifically, If *func_ptr* is a function pointer stored in a data-structure field, we get its candidate functions according to its data-structure type, field offset and function-pointer type. Thirdly, our analysis selects the functions from *cand_func_set* whose source file *cand_src_file* has a link-information connection with *src_file* (lines 4-9). The selected functions are put in *func_set*. Because a link-information connection is strong, if any function is selected through this connection, our analysis returns *func_set* and ends (lines 10-12). Otherwise, our analysis selects the functions from *cand_func_set* whose source file

```
FuncPtrAnalysis(func_ptr_call, src_file)
Input: func_ptr_call - function pointer call;
       src_file - name of the source file where summary-based analysis starts
Output: func_set - set of the functions referenced by the function pointer
 1: func_set := ø;
 2: func_ptr := GetCalledValue(func_ptr_call);
 3: cand_func_set := FindCandidateFuncSet(func_ptr);
 4: foreach cand_func in cand_func_set do
 5:     cand_src_file := GetSourceFile(cand_func);
 6:     if HaveLinkInfoConnection(cand_src_file, src_file) == TRUE then
 7:         AddFuncSet(cand_func, func_set);
 8:     end if
 9: end foreach
10: if func_set != ø then
11:     return func_set;
12: end
13: foreach cand_func in cand_func_set do
14:     cand_src_file := GetSourceFile(cand_func);
15:     if HaveFuncCallConnection(cand_src_file, src_file) == TRUE then
16:         AddFuncSet(cand_func, func_set);
17:     end if
18: end foreach
19: return func_set;
```

Fig. 8. Procedure of connection-based function-pointer analysis.

cand_src_file has a function-call connection with *src_file* (lines 13-18). The selected functions are put in *func_set*. Finally, our analysis returns *func_set* and ends (line 19).

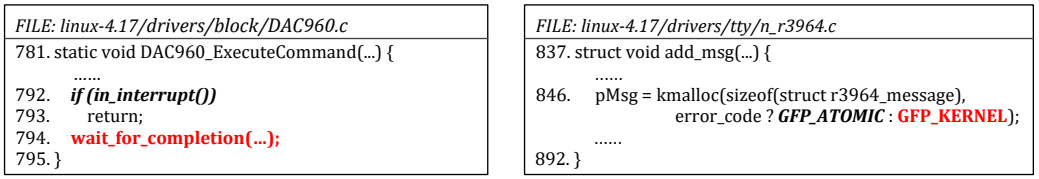
The main advantage of our connection-based alias analysis is to improve the accuracy of identifying correct referenced functions for function-pointer calls. This advantage can benefit the detection of SAC bugs in two aspects: (1) Deep SAC bugs involving function pointers can be detected. (2) Filtering out incorrect functions for function pointers reduces the false positive rate. Another advantage of our alias analysis is high efficiency, because the analysis is flow-insensitive and its processing is not complex. This advantage is quite beneficial in analyzing code of millions of lines, as found in the Linux kernel. However, due to flow insensitivity, the identified function(s) may be incorrect in a specific control flow. Moreover, in the current implementation, our analysis focuses on function pointers stored in data-structure fields, and other kinds of function pointers are not handled, so some SAC bugs involving these unhandled function pointers may be missed.

At present, our connection-based alias analysis is only used to detect SAC bugs. In another work [5], we have developed the tool DCNS that detects the inverse of SAC bugs, where non-sleep operations are used unnecessarily, outside of atomic context. That work uses a simpler version of connection-based alias analysis that only considers function call connections and does not take into account link information. In both of these cases, our alias analysis only handles function-pointer calls that are considered to occur in atomic context. We are currently extending our alias analysis to detect other kinds of bugs involving function-pointer calls.

4.3 Path-Check Bug Filtering

Given the code paths recorded during our summary-based analysis, we use a path-check method to filter out repeated reports and false bugs.

Firstly, our method filters out repeated bugs. For each new possible bug, our method checks whether its starting and ending basic blocks are the same as those of an already detected bug, and whether the ending basic block uses the same sleep-able function call. If the two conditions are both satisfied, the possible bug is marked as a repeated bug and is filtered out.



(a) Specific kernel interface. (b) Non-sleep constant flag.

Fig. 9. Examples of code styles for avoiding SAC bugs.

Secondly, our method filters out false bugs, which are mainly introduced by the fact that our summary-based analysis neglects variable value information. Our basic strategy is to validate the path conditions [12] of the code path for each possible bug. As shown in Figure 4, the code path consists of nodes, and each node can be a basic block or a function call whose called function is analyzed. Our method scans each node in the code path in order, records the stored value of each variable in each basic block, and validates the branch condition of each basic block according to the recorded values of related variables. If the validation fails, the possible bug is marked as a false bug and is filtered out.

Besides, by studying the Linux kernel code, we also find two common coding styles used by kernel developers to avoid introducing SAC bugs: (1) A function call to a specific kernel interface (like `in_interrupt`) is used to check for atomic context. (2) A non-sleep constant flag (like `GFP_ATOMIC`) is used as an argument under a condition in the function call. Figure 9 shows two examples in the Linux kernel code. In Figure 9(a), a conditional tests the result of calling `in_interrupt` to check whether the code is executed in an interrupt handler and decide whether the sleep-able function `wait_for_completion` can be called. In Figure 9(b), the call to the function `kmalloc` chooses between using a non-sleep constant flag `GFP_ATOMIC` or a sleep-able constant flag `GFP_KERNEL` as an argument. In fact, these two coding styles indicate semantic information, namely that the code can be in atomic context or non-atomic context. If a code path associated with a possible bug satisfies either of these two code styles, the possible bug is marked as a false bug and is filtered out.

5 APPROACH

Based on the three techniques in Section 4, we propose a static approach DSAC, to effectively detect SAC bugs in the Linux kernel. We have implemented DSAC using the Clang compiler [16], and perform static analysis on the LLVM bytecode of each source file. Figure 10 shows the architecture of DSAC, which has five parts:

- ***Code compiler.*** This part compiles each source file of the Linux kernel into an LLVM bytecode file, and records link information during code compilation and linking.
- ***Information collector.*** This part analyzes each LLVM bytecode file to collect useful information about the code, and stores the information in a MySQL database [43]. This information will be used in the remaining analyses.
- ***Connection extractor.*** This part uses the collected code information and link information to extract connections between source files, and stores them in the MySQL database.
- ***Bug detector.*** This part performs our summary-based analysis to detect possible SAC bugs from the LLVM bytecode.
- ***Bug filter.*** This part uses our path-check method to filter out repeated and false bugs, and generates reports for the final detected SAC bugs.

Based on this architecture, DSAC consists of four phases, which will be introduced in Sections 5.1-5.4. Section 5.5 will analyze the parallelism of these phases.

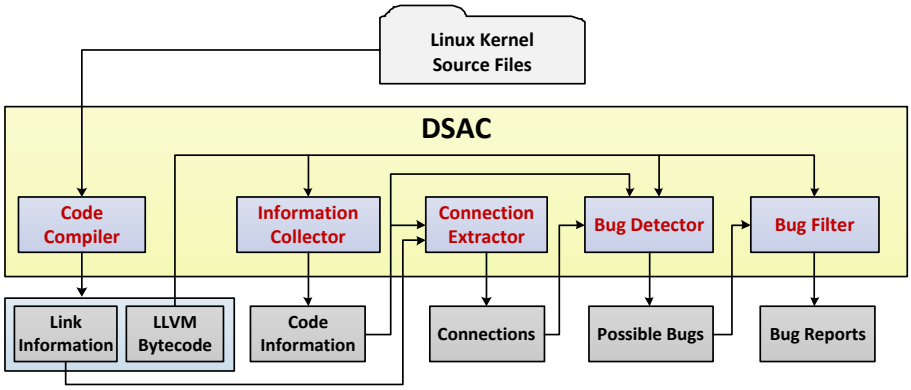


Fig. 10. DSAC architecture.

5.1 Code Compilation

In this phase, the code compiler performs code compilation and linking using the Clang compiler, in three steps: (1) compile each source file into a LLVM bytecode file; (2) compile each bytecode file into an assembly file and then into an object file; (3) link one or more object files together into a kernel object file, representing a loadable kernel module.

The code compiler keeps the set of LLVM bytecode files generated in the first step for the remaining analyses, and records the link information obtained from the third step.

5.2 Information Collection

In this phase, by analyzing LLVM bytecode files, the information collector extracts and stores useful information about the code of the Linux kernel, including the name and position of each function definition, the functions that may be referenced by each function pointer, and the callee and caller functions of each direct function call.

5.3 Connection Extraction

In this phase, the connection extractor extracts connections between source files. To extract link-information connections, the extractor analyzes the recorded link information, to get the source files that are linked together. To extract function-call connections, the extractor analyzes each function call between two different source files, and records the information that these two source files have a direct function-call connection.

5.4 Bug Detection

In this phase, the bug detector first performs summary-based analysis on the LLVM bytecode files, using the collected code information. To handle function-pointer calls, the detector performs connection-based alias analysis to identify the set of functions referenced by the function pointer, and processes each function in the resulting set. Then, the bug filter filters out repeated reports and false bugs. Finally, DSAC produces detailed reports for the found SAC bugs (including code paths and source file names), to help the user to locate the bugs.

5.5 Parallelism

The first three phases each work on the individual files independently, benefiting from the information collected in the previous phases. Thus, these phases can be parallelized straightforwardly.

Table 1. Evaluated Linux kernel versions

Description	Linux 3.17.2	Linux 4.17
Release time	October 2014	June 2018
Source files(.c)	22.8K	25.9K
Source code lines	12.4M	16.9M

In the fourth phase, the summary-based analysis traces through function calls, which may cross file boundaries, when the called function is defined in another file and no function summary is available for the current analysis. In this case, we can also run the analysis in parallel, relying on the global database of function summaries to prevent repeated analysis of functions in the same context across different threads.

6 EVALUATION

6.1 Experimental Setup

To validate the effectiveness of DSAC, we evaluate it on the Linux kernel code. To cover different kernel versions, we select an old version 3.17.2 (released in October 2014) and a recent version 4.17 (released in June 2018). Table 1 shows some information about these kernel versions (the lines of source code are measured with CLOC [17]). We run the experiments on a Lenovo x86-64 PC with four Intel i5-3470@3.20G processors and 8GB memory. We compile the kernel code using Clang 6.0. We use the kernel configuration `allyesconfig` to enable all code for the x86 architecture. Because DSAC can work in parallel, we configure DSAC to run on four threads.

To run DSAC, we perform three steps. Firstly, we configure DSAC for the Linux kernel, by providing the names of 29 pairs of spin-lock and spin-unlock functions (such as `spin_lock_irq` and `spin_unlock_irq`), 3 interrupt-handler-register functions (`request_irq`, `tasklet_init` and `devm_request_irq`), 24 basic sleep-able kernel interfaces (such as `msleep` and `usleep_range`) and 5 sleep-able flags (such as `GFP_KERNEL and GFP_USER`). Secondly, we execute DSAC's compiling script to compile the source code of the Linux kernel. Finally, we execute DSAC's bug-detection script to detect SAC bugs. The second and third steps are fully automated.

6.2 Bug Detection

To validate whether DSAC can find known bugs, we use DSAC to check the Linux 3.17.2 kernel source code. To validate whether DSAC can find new bugs, we use DSAC to check the Linux 4.17 kernel source code. To check the accuracy of the bug-detection results, we manually check all detected bugs to identify whether they are real bugs. Table 2 shows the results. In this table, column "DSAC" shows the results of DSAC, and column "DSAC_noptr" shows the results of a variant of DSAC that does not consider the sets of functions possibly invoked by function-pointer calls. From Table 2, we have the following findings:

(1) DSAC can scale to large code bases. In Linux 3.17.2 and 4.17, DSAC respectively handles the 11.7K and 16.7K source files included in the x86 `allyesconfig` Linux kernel configuration. These files contain 7.0M and 9.8M lines of source code, as measured with CLOC [17]. DSAC starts its analysis from many entry basic blocks and many interrupt handler (INTR) functions. Still, some source files of the Linux kernel are not handled by DSAC, because they are not enabled for the x86 architecture.

(2) DSAC effectively handles many function-pointer calls, namely it successfully identifies the referenced functions of many function-pointer calls. DSAC respectively handles 73% and 67% of all encountered function-pointer calls in Linux 3.17.2 and 4.17. Among these handled function-pointer calls, 85% and 83%, respectively, are handled by link-information connections.

Table 2. Bug detection results.

Description		Linux 3.17.2		Linux 4.17	
		DSAC	DSAC_noptr	DSAC	DSAC_noptr
Code handling	Source files (.c)	11.7K	11.7K	16.7K	16.7K
	Source code lines	7.0M	7.0M	9.8M	9.8M
Summary-based analysis	Handled INTR functions	966	966	1424	1424
	Entry basic blocks	42190	42190	50929	50929
	Handled functions	51K	37K	65K	47K
	Function summaries	79K	52K	103K	69K
Function-pointer analysis	Encountered function-pointer calls	14185	-	17349	-
	Handled function-pointer calls	10323	-	11915	-
	Calls handled by link-information connection	8741	-	9859	-
	Calls handled by function-call connection	1582	-	2056	-
	Candidate referenced functions	113K	-	138K	-
	Identified referenced functions	40K	-	45K	-
Bug detection	Filtered repeated bugs	22687	5151	28374	11586
	Filtered false bugs	7796	4284	9957	5141
	Final detected bugs	891	464	1159	615
	Real bugs	805	432	1068	564
Time usage	P1: Information collection	17m	17m	24m	24m
	P2: Connection extraction	8m	0m	11m	0m
	P3: Bug detection	53m	23m	62m	28m
	Total	78m	40m	97m	52m

FILE: linux-4.17/fs/lockd/svc4proc.c

```
278. static __be32 nlm4svc_callback(..., __be32(*func)(...)) {
        ......
294.    if (call == NULL)
295.        return rpc_system_err;
296.
297.    stat = func(...);
        ......
307. }
-------------------------------------------------------------
310. static __be32 nlm4svc_proc_test_msg(...) {
311.    dprintk("locked: TEST_MSG called\n");
312.    return nlm4svc_callback(..., __nlm4svc_proc_test);
313. }
```

(a) Function pointer of function argument.

FILE: linux-4.17/net/atm/raw.c

```
78. int atm_init_aal5(...) {
        ......
83.    vcc->send = vcc->dev->ops->send;
84.    return 0;
85. }
```

FILE: linux-4.17/net/atm/clip.c

```
327. static netdev_tx_t clip_start_xmit(...) {
        ......
395.    vcc->send(...);
        ......
413. }
```

(b) Function pointer of indirect assignment.

Fig. 11. Examples of unhandled function-pointer calls by DSAC.

The remaining 27% and 33% of function-pointer calls are not handled. There are two cases that are commonly not covered. Firstly, DSAC focuses on function pointers stored in data-structure fields, and cannot handle the function pointers that are not referenced in this way. For example in Figure 11(a), on line 312, the function `nlm4svc_proc_test_msg` calls the function `nlm4svc_callback` with an argument that is a pointer to the function `__nlm4svc_proc_test`. And the function `nlm4svc_callback` calls `__nlm4svc_proc_test` via the pointer argument on line 297. Because the function pointer on line 297 is not a data-structure field, DSAC cannot identify the referenced functions for this function pointer call. Secondly, some function pointers stored in data-structure fields are not directly assigned to explicit functions, and DSAC cannot identify the functions involved in such function-pointer assignments. For example in Figure 11(b), the function pointer `vcc->send` is assigned through another function pointer `vcc->dev->ops->send` on line 83, so the candidate referenced functions of `vcc->send` cannot be directly identified only according to this assignment. For this reason, DSAC cannot identify the referenced functions for the function-pointer call to `vcc->send` on line 395.

(3) Our connection-based alias analysis is effective in improving the accuracy of function-pointer analysis. It retains around 35% and 33% of the candidate referenced functions in Linux 3.17.2 and 4.17, respectively. The remaining 65% and 67% candidate referenced functions are discarded because they are considered to be incorrect in the calling context of the given function-pointer call. By discarding these incorrect functions, DSAC reduces unnecessary analysis of atomic context, and also reduces the number of false positives in bug detection.

(4) Our path-check method is effective in filtering out repeated reports and false bugs. It filters out more than 95% of all possible bugs, which are considered to be repeated reports and false positives according to their code paths. Among the false bugs filtered out by DSAC, 28% and 25% are related to coding styles about atomic context (such as using the function call to `in_interrupt` and the non-sleep constant flag `GFP_ATOMIC`).

(5) DSAC is effective in bug detection with or without function-pointer analysis:

For Linux 3.17.2, DSAC reports 891 bugs, of which we have identified 805 as real bugs. Two experienced researchers each spent six days manually checking these bugs. Among these bugs, 304 bugs have been fixed in Linux 4.17. Without function-pointer analysis (*DSAC_noptr*), it reports 464 bugs, of which we have identified 432 as real bugs. Among these bugs, 171 bugs have been fixed in Linux 4.17. The results indicate that DSAC can indeed find known bugs.

For Linux 4.17, DSAC reports 1159 bugs, of which we have identified 1068 as real bugs. Two experienced researchers each spent eight days manually checking these bugs. Without function-pointer analysis (*DSAC_noptr*), it reports 615 bugs, of which we have identified 564 bugs as real bugs. Among the 1068 real bugs found by DSAC, we randomly selected 300 and reported them to kernel developers. As of September 2018, 220 bugs have been confirmed. We also sent 94 patches to fix 208 of the reported bugs, and 51 patches fixing 115 bugs have been applied by the kernel maintainers. The results indicate that DSAC can indeed find new bugs.

(6) Using function-pointer analysis, DSAC can find many deep bugs that involve function-pointer calls. In Linux 3.17.2 and 4.17, DSAC respectively finds 341 and 505 more real bugs than that without function pointer analysis (*DSAC_noptr*), accounting for 42% and 47% of all real found bugs.

(7) DSAC achieves good accuracy in bug detection. The false positive rates are respectively only 9.7% and 7.9% in Linux 3.17.2 and 4.17. Without function-pointer analysis (*DSAC_noptr*), the false positive rates are respectively only 6.9% and 8.3%.

(8) DSAC is efficient in code analysis. On four running threads, it respectively spends 78 and 97 minutes on handling 11.7K and 16.7K source files in Linux 3.17.2 and 4.17. Without function-pointer analysis, on four running threads, DSAC respectively spends 40 and 52 minutes of running time.

6.3 Example Bugs

Figure 12 shows some real bugs found by DSAC in Linux 4.17, that have been confirmed by kernel developers. In Figure 12(a), the *socionext* driver registers the function `ave_irq_handler` as an interrupt handler using the kernel interface `request_irq` on line 1236. The function `ave_irq_handler` calls `ave_rxfifo_reset` on line 932, which calls a sleep-able function `usleep_range` on lines 888 and 892, causing two SAC bugs.[3]

In Figure 12(b), the function `__setup_irq` calls `raw_spin_lock_irqsave` to acquire a spinlock, and then calls `__irq_set_trigger` on line 1353. This called function uses a function pointer call `chip->irq_set_type` on line 685. In the *adp5588* driver, this function pointer is assigned to the function `adp5588_irq_set_type` on line 237. This function calls `adp5588_gpio_direction_input` on line 224, which calls a sleep-able function `mutex_lock` on line 113, causing a SAC bug.[4]

[3]Link: https://lore.kernel.org/patchwork/patch/986431/

[4]Link: https://lkml.org/lkml/2018/8/13/197

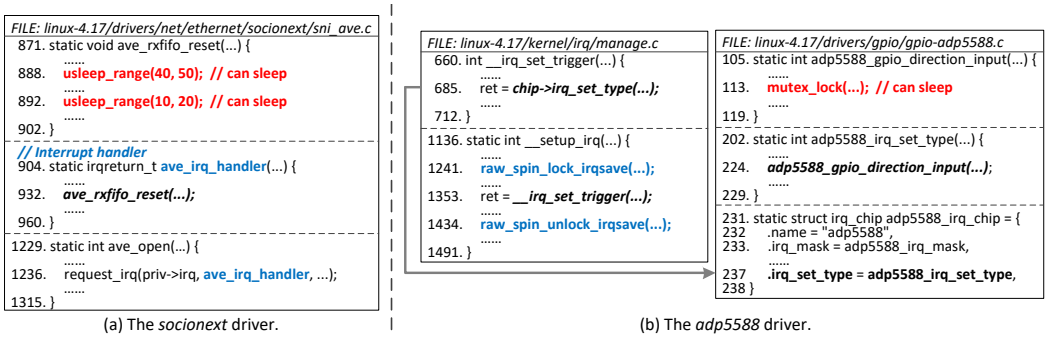


Fig. 12. Examples of real bugs found by DSAC.

Figure 13 shows the bug reports for these bugs generated by DSAC. The report explicitly shows the function call path (from bottom to top) of the detected bug, including the source file's name, the source line number, the callee function's name and the caller function's name. In the report, "[FUNC_PTR]" means that a function-pointer call is used. According to the bug report, the user can conveniently locate the detected bug.

```
========== BUG [INTR] ==========
[FUNC] usleep_range
drivers/net/ethernet/socionext/sni_ave.c, 888: usleep_range in ave_rxfifo_reset
drivers/net/ethernet/socionext/sni_ave.c, 932: ave_rxfifo_reset in ave_irq_handler

......
========== BUG ==========
[FUNC] mutex_lock
drivers/gpio/gpio-adp5588.c, 113: mutex_lock in adp5588_gpio_direction_input
drivers/gpio/gpio-adp5588.c, 224: adp5588_gpio_direction_input in adp5588_irq_set_type
kernel/irq/manage.c, 685: [FUNC_PTR]adp5588_irq_set_type in __irq_set_trigger
kernel/irq/manage.c, 1353: __irq_set_trigger in __setup_irq
kernel/irq/manage.c, 1241: raw_spin_lock_irqsave in __setup_irq

......
```

Fig. 13. Example of bug reports generated by DSAC.

6.4 Bug Characteristics

Reviewing the SAC bugs found by DSAC, we find some interesting characteristics of them:

(1) Most of the detected bugs (1056 in Linux 4.17) involve multiple function calls. Indeed, kernel developers may easily forget that the code is in atomic context across multiple function calls, especially function-pointer calls.

(2) Many of the detected bugs (313 in Linux 4.17) involve multiple parts of the kernel. For example in Figure 12(b), the SAC bug involves two parts of the kernel, namely the interrupt handling manager and the driver *adp5588*. Figure 14 shows another representative bug found by DSAC and the generated bug report. This SAC bug involves three drivers of different classes, namely *input*, *mfd* and *spi* drivers. Indeed, when calling functions defined in other parts of the kernel, kernel developers may easily forget to check whether these functions can sleep and just call these functions according to their functionality.

(3) Many of the detected bugs (235 in Linux 4.17) are related to resource allocation. For example, many SAC bugs are caused by allocating a resource using a sleep-able flag `GFP_KERNEL` (like the bugs shown in Figure 4), which allows sleeping and waiting until the resource becomes available.

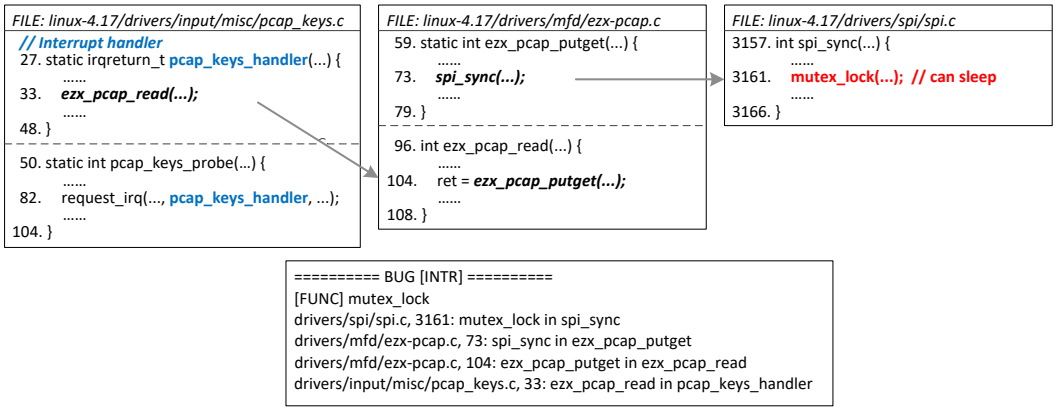


Fig. 14. Example SAC bug involving multiple parts of the kernel and its bug report.

(4) Multiple detected bugs are caused by calling the same sleep-able kernel interfaces. This characteristic indicates that some commonly used kernel interfaces may be supposed to be non-sleep by some kernel developers, but these kernel interfaces can sleep in reality. An example is that the three bugs shown in Figure 4 are all caused by calling the sleep-able kernel interface `drm_mm_takedown`. Besides, in DSAC's bug reports, `hid_hw_request` (causing 18 bugs) and `clk_get_rate` (causing 6 bugs) are also such sleep-able kernel interfaces.

(5) Few of the detected bugs (152 in Linux 4.17) are in interrupt handlers. Indeed, driver developers often write clear comments to mark the driver functions that are called from an interrupt handler, to protect against calling sleep-able functions in these functions.

6.5 Bug Distribution

We classify the detected bugs according to the directory of their source files. As described in Section 6.4, many detected bugs involve multiple parts of the kernel, so the bugs are classified by the source file of the starting basic block (containing the call to a spin-lock function or the entry of an interrupt handler function) in the bug report. Figure 15 shows the results for the real bugs identified by our manual check.

We find that drivers have more than 77% of all detected real bugs. This is somewhat higher than the percentage of code represented by drivers (67% in Linux 4.17). Thus, drivers remain a significant source of system problems [52]. We also classify the detected bugs in drivers by the driver class. We find that *network*, *scsi* and *staging* drivers together have more than 50% of the bugs in drivers. A possible reason is that these drivers also have much more code than the other driver classes.

6.6 False Positives

False positives are mainly introduced in two cases:

Firstly, due to flow insensitivity, our connection-based alias analysis may identify incorrect referenced functions for function pointers. Figure 16(a) shows a false reported bug in such a case. The function `qla2100_intr_handler` acquires a spinlock on line 59, and then uses a function-pointer call `ha->isp_ops->fw_dump` on line 77. This function pointer has several candidate referenced functions, such as `qla2100_fw_dump` and `qla8044_fw_dump`. Because the two functions are respectively defined in *qla_dbg.c* and *qla_nx2.c*, which are both linked with *qla_isr.c* and *qla_os.c* according to the Makefile, our alias analysis identifies both of the functions as referenced functions of the function-pointer call. But according to the data flow of the source file, `qla2100_fw_dump` should

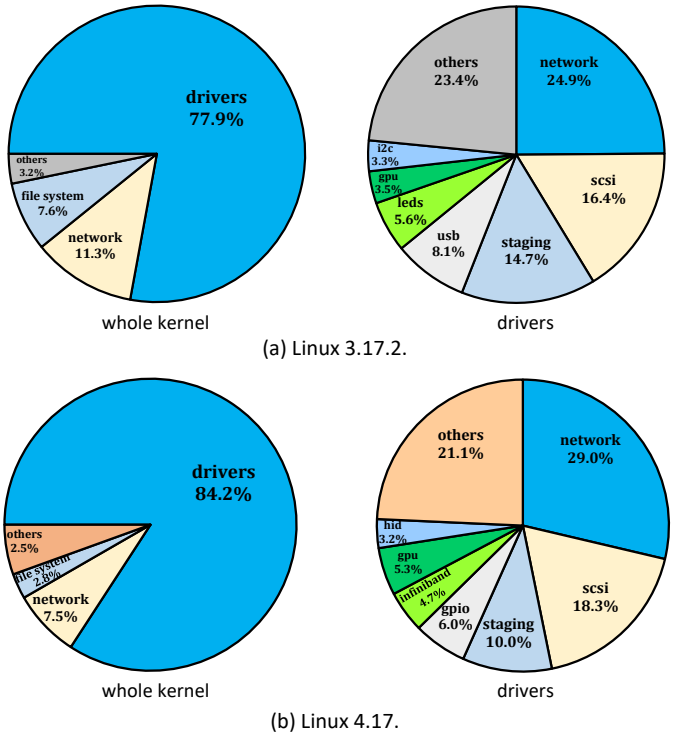


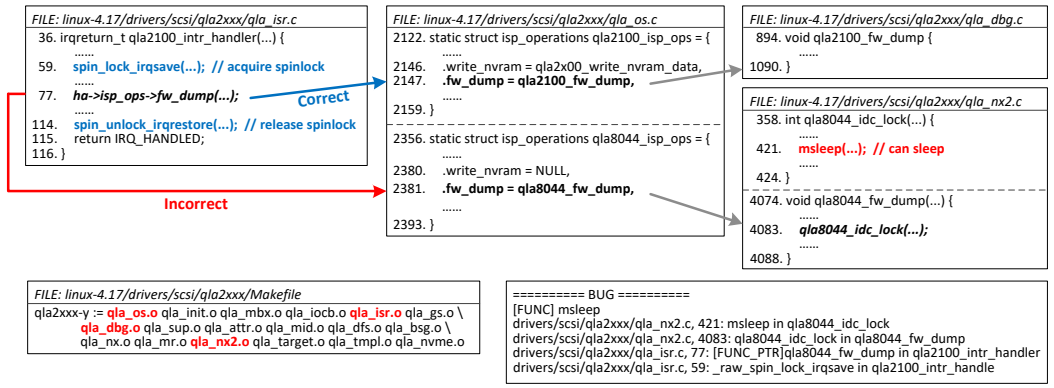
Fig. 15. The distribution of real SAC bugs found by DSAC.

be the sole correct referenced function of the function-pointer call. Thus, the detected bug about `msleep` on line 421 called by `qla8044_fw_dump` through `qla8044_idc_lock` is a false positive.

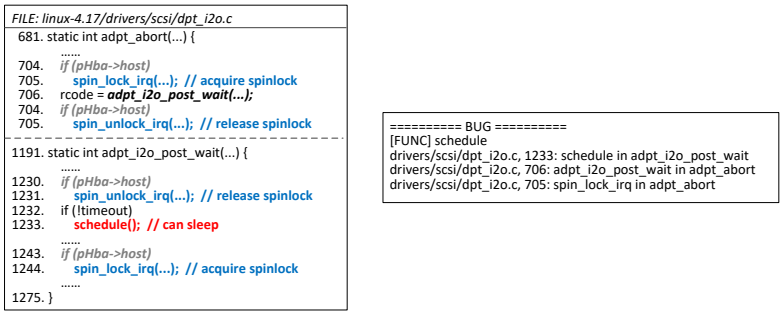
Secondly, our path-check method fails to identify infeasible code paths in some complex cases. The case of acquiring a spinlock under a condition and checking it across function calls is an example, and Figure 16(b) shows a false reported bug in such case. The function `adpt_abort` acquires a spinlock on line 705 under the condition that `pHba->host` is *true* on line 704, and then calls the function `adpt_i2o_post_wait` on line 706. The function `adpt_i2o_post_wait` releases the spinlock on line 1231 under a condition that `pHba->host` is *true* on line 1230, and then calls a sleep-able function `schedule` on line 1233. Thus, `schedule` is not called while holding the spinlock. However, DSAC does not maintain the condition `pHba->host` on line 704 across function calls, because this variable is a data-structure field, which the path-check method does not handle at present. Thus, our path-check method cannot use this condition to validate the path feasibility, and considers that `schedule` can be called while holding the spinlock. Thus, it reports a false bug.

6.7 False Negatives

To analyze the false negatives of DSAC, we compare its bug reports to the kernel commits fixing SAC bugs by changing `GFP_KERNEL` into `GFP_ATOMIC` between Linux 4.17 and 5.2 (June 2018 - July 2019). Specifically, we collect the commits that were not submitted by us and where the bug was already present in Linux v4.17. This strategy resulted in 16 commits. DSAC finds the bugs in 5 of these commits, but misses the bugs in the remaining 11 commits. These bugs are missed for two main reasons:



(a) Incorrect referenced function of function pointer call.



(b) Infeasible code path.

Fig. 16. Example of false detected bugs and their bug reports.

Firstly, our connection-based alias analysis focuses on direct function pointer assignments involving data-structure fields, but neglects other kinds of function-pointer assignments. For example, a function pointer may be assigned through a function argument or another function pointer (as illustrated by the examples in Figure 11). Thus, our alias analysis fails to identify the set of functions referenced by calls to such a function pointer, which may cause DSAC to miss real bugs involving these calls. This reason causes DSAC to miss the bugs in 7 commits.

Secondly, DSAC requires the kernel configuration, and we only use the `allyesconfig` configuration for the x86 architecture in our evaluation. Thus, SAC bugs in the source files that are not enabled by this configuration are missed. This reason causes DSAC to miss the bugs in 4 commits.

6.8 Common Fixing Patterns for SAC Bugs

By studying Linux kernel patches, we have found four common patterns for fixing SAC bugs. In Figure 17, we illustrate each fixing pattern using a known SAC bug in Linux 3.17.2 that has been fixed in Linux 4.17.

The fixing patterns are as follows:

P1: Replace the sleep-able function call with a non-sleep function call having the same functionality, as illustrated by the change `usleep_range` ⇒ `udelay` shown in Figure 17(a).[5]

[5]Patch link: https://github.com/torvalds/linux/commit/69b624983f94f2a877449c1e6c34f21c97440f25

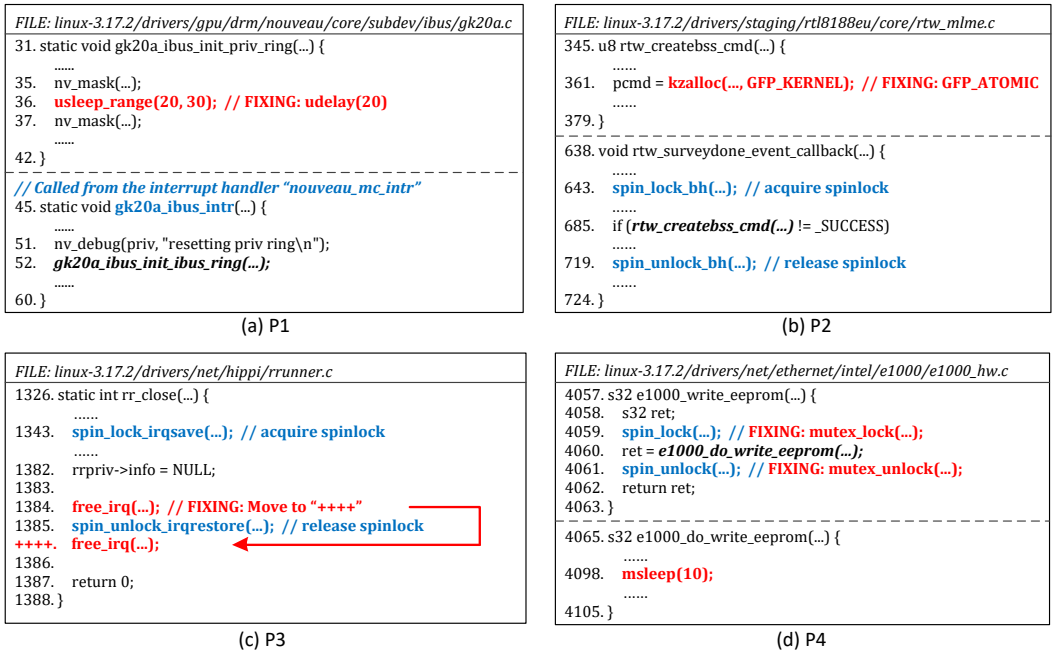


Fig. 17. Examples of common fixing patterns for SAC bugs in Linux 3.17.2.

P2: Replace the specific sleep-able constant flag with a non-sleep flag, as illustrated by the change `GFP_KERNEL` ⇒ `GFP_ATOMIC` shown in Figure 17(b).[6]

P3: Move the sleep-able function call to some place where a spinlock is not held, as illustrated by the change of moving `free_irq` shown in Figure 17(c).[7]

P4: Replace the spinlock with a lock that allows sleeping, as illustrated by the change of `spin_lock` ⇒ `mutex_lock` and `spin_unlock` ⇒ `mutex_unlock` shown in Figure 17(d).[8]

These patterns have different usage scenarios and raise different challenges. Firstly, P1 and P2 can be used for all atomic contexts, while P3 and P4 are only used when holding a spinlock. Secondly, P1 and P2 involve simple modifications, while P3 and P4 involve more difficult modifications and are error-prone. Using P3 requires carefully determining where the sleep-able function should be moved to. Using P4 requires modifying all locking and unlocking operations. Thus, when we write our patches for fixing the detected bugs in Linux 4.17, we preferentially select the bugs that can be fixed using P1 and P2.

6.9 Sensitivity Analysis

DSAC exploits two key techniques: a summary-based analysis to reduce repeated analysis, and a connection-based alias analysis to identify the set of functions referenced by a function pointer. To better understand the value of these two techniques, we modify DSAC to drop each of them, and evaluate each modified tool on Linux 3.17.2.

Summary-based analysis. We implement a modified tool by dropping the summary-based analysis, implying that functions may be repeatedly analyzed in the same execution context. The

[6]Patch link: https://lkml.org/lkml/2017/12/18/833
[7]Patch link: https://patchwork.ozlabs.org/patch/847407/
[8]Patch link: https://patchwork.ozlabs.org/patch/499802/

PathHasExisted check is retained, however, on lines 1-3 of *HandleBlock* (Figure 3), to protect against infinite loops due to recursion. In this experiment, the modified tool runs for over 11 hours and finally aborts due to insufficient memory, without completing the analysis. This experiment indicates that our summary-based analysis indeed improves the efficiency of atomic context analysis.

Connection-based alias analysis. We implement a modified tool by dropping the connection-based alias analysis, and simply using a field-based analysis [30]. Specifically, we simply select all possible referenced functions of a function pointer stored in a data-structure field, according to its data-structure type, field offset and function-pointer type. In the experiment, the modified tool runs for around 320 minutes, and identifies more than 136K referenced functions of function-pointer calls, which is much more than the number of referenced functions (40K) identified by original DSAC. Because these referenced functions have to be all handled in the code analysis, the time usage of the modified tool is much longer than that of original DSAC (320 minutes vs. 78 minutes, *i.e.*, over 4x longer). As for bug detection, the modified tool reports 2958 SAC bugs, much more than the number of bugs found by original DSAC. We have manually checked these reported bugs, and found that more than 2000 of them are false. The reason is that in the recorded code paths of these false bugs, the referenced functions of function-pointer calls are incorrect. This experiment indicates that our connection-based alias analysis indeed reduces the false positive rate of SAC bug detection and improves the efficiency of atomic context analysis.

7 COMPARISON TO PREVIOUS APPROACHES

Several previous approaches [4, 7, 15, 24, 46] have considered SAC bugs. Among them, we select two state-of-the-art approaches, namely the Coccinelle BlockLock checker [46] and our previous DSAC approach [7] to make detailed comparisons. We select the Coccinelle BlockLock checker because it has been used to detect many SAC bugs in the Linux kernel and it is open-source and its bug reports are available [11]. We select the previous DSAC approach because we aim to show the improvements of the current DSAC approach in this paper.

7.1 Coccinelle BlockLock Checker

Compared to the BlockLock checker, DSAC has some important improvements:

Atomic context analysis. BlockLock only uses one bit of context information to check if a lock is held, so it may not accurately identify the code in atomic context when multiple locks are taken but only some of them are released. DSAC maintains a complete lock stack during its summary-based analysis, thus it can be more accurate in identifying the code in atomic context. BlockLock is also not sensitive to the module Makefile, and thus may choose the wrong definition when unfolding a function call if the called function has multiple definitions. DSAC uses the module Makefile to accurately identify the definition of each function. DSAC can detect SAC bugs in interrupt handlers and involving sleep-able operations other than a call to an allocation function with `GFP_KERNEL`, which are not considered by BlockLock.

Function-pointer analysis. BlockLock does not handle function-pointer calls. DSAC uses a connection-based alias analysis to handle function-pointer calls. Thus, DSAC can find the bugs involving function-pointer calls, which are missed by BlockLock.

False bug filtering. BlockLock does not consider variable value information to validate path conditions, which may cause a number of false positives. DSAC uses a path-check method to validate whether the code path is feasible, which filters out many false bugs.

We also compare the experimental results of BlockLock and DSAC, with two steps. Firstly, we download the bug reports of BlockLock for Linux 2.6.33, and get 70 reported bugs. We select the bugs related to the x86 architecture based on the Kconfig files, leaving 37 reported SAC bugs (26 real bugs and 11 false bugs). Secondly, we use DSAC to check the source code of Linux 2.6.33. We

use the kernel configuration `allyesconfig` to enable all code for the x86 architecture. DSAC runs for 45m on four running threads, and finds 772 SAC bugs. We manually check these bugs, and find that 719 are real.

By manually comparing the bug reports, we find that: (1) 59 real bugs reported by DSAC are equivalent to 26 real bugs reported by BlockLock. DSAC reports more bugs because it detects basic sleep-able kernel interfaces, while BlockLock detects sleep-able functions. Thus, if a function defined in the kernel module calls several basic sleep-able kernel interfaces in atomic context, DSAC reports all these kernel interfaces, while BlockLock only reports this function. (2) DSAC filters out all false bugs reported by BlockLock. (3) DSAC reports 660 real bugs missed by BlockLock. Some of these bugs involve multiple source files, and BlockLock cannot handle them very precisely; some of these bugs involve function-pointer calls or are related to interrupt handling, which are not considered by BlockLock. (4) The false positive rate of DSAC is 6.9%, which is lower than that of BlockLock.

However, compared to BlockLock, an important limitation of DSAC is that its results are specific to a single kernel configuration. BlockLock is based on Coccinelle [45], which does not compile the source code. Thus it can conveniently check all source files without any kernel configuration. DSAC is based on LLVM, which compiles the source code with a selected kernel configuration. Thus, the 33 bugs found by BlockLock for non-x86 architectures are missed by DSAC.

7.2 Our Previous DSAC approach

Compared to our previous DSAC approach [7], our current DSAC approach achieves some important improvements:

Summary-based analysis. Our previous DSAC approach uses a hybrid of flow-sensitive and -insensitive analysis, which may repeatedly analyze a function under the same execution context. Our current DSAC approach uses a summary-based analysis, which only analyzes a function once per execution context. And our current DSAC approach uses a flow-sensitive analysis to analyze all targeted code paths. Thus, our current DSAC approach is more efficient and accurate than our previous DSAC approach.

Cross-kernel-module analysis. Our previous DSAC approach handles one kernel module at a time and only analyzes code paths within the module. Our current DSAC approach can handle multiple kernel modules and can analyze code paths across different kernel modules. Thus, our current DSAC approach can find bugs involving multiple kernel modules that are missed by our previous DSAC approach.

Connection-based alias analysis. Our previous DSAC approach does not handle function-pointer calls. Our current DSAC approach uses a connection-based alias analysis to identify the set of functions referenced by a function pointer. Thus, our current DSAC approach can find many bugs involving function-pointer calls that are missed by our previous DSAC approach.

Improved path-check method. Our previous DSAC approach relied on some information about common coding styles to check the names of variables and called functions in *if* conditions. This approach is only applicable to the cases that follow the expected coding styles, and thus many false bugs involving other cases are still reported. Our current DSAC approach collects some information about variable values to validate the feasibility of the code path, which is more accurate and effective in filtering out false bugs.

Our previous DSAC and current DSAC approaches are both evaluated on Linux 3.17.2. However, our previous DSAC approach only checks device drivers, while our current DSAC approach checks the whole kernel. To make a fair comparison, we select the SAC bugs of drivers found by our current DSAC approach, and get 666 bug reports (627 real bugs and 39 false bugs). Our previous DSAC approach reports 215 bugs in drivers (200 real bugs and 15 false bugs). By manually checking

the bug reports, we find that: (1) The 200 real bugs found by our previous DSAC approach are all found by our current DSAC approach. (2) Among the 15 false bugs found by our previous DSAC approach, 10 are filtered out by our current DSAC approach. (3) Our current DSAC approach finds 427 real bugs missed by our previous DSAC approach. These bugs involve multiple kernel modules or function-pointer calls, and our previous DSAC approach cannot handle them. (4) The false positive rate of our current DSAC approach is 5.9%, which is lower than that of our previous DSAC approach (7.0%).

8 DISCUSSION

In this section, we discuss two possible extensions for DSAC.

8.1 Detecting SAC Bugs in Other OS Kernels

At present, DSAC is specific to the Linux kernel, in two main aspects. Firstly, DSAC requires LLVM bytecode and link information for the OS kernel, and we implement a script specific to the Linux kernel to collect this information when compiling the Linux kernel source code. Secondly, DSAC requires the names of some functions and flags specific to the Linux kernel. For the summary-based analysis in Figure 3, we need to manually provide the names of the spin-lock and spin-unlock functions used on lines 6 and 8 in *HandleBlock* and on line 3 in *CodeAnalysis*, the names of the interrupt-handler-register functions used on line 8 in *CodeAnalysis*, and the names of the basic sleep-able kernel interfaces and sleep-able flags used on lines 10-11 in *CodeAnalysis*. For the path-check method, we need to manually provide the names of the functions checking atomic context and the non-sleep flags. Thus, to apply DSAC in another OS kernel, such as FreeBSD or NetBSD, we need to implement a new script for the source-code compilation of this kernel, and provide the names of the corresponding functions and flags specific to the targeted kernel.

8.2 Detecting Other Kernel Problems

The good results and reasonable performance of DSAC rely on three key techniques used in DSAC: function summaries to avoid repeated analysis, link and file connections to reduce the set of considered functions when analyzing function-pointer calls, and a path-check method that only applies precise analysis to possible bug reports to avoid the cost of always checking during static analysis. DSAC also benefits from the fact that it is looking for a specific bug type, involving specific operators, such as locking calls and sleep-able functions, that occur frequently, but not pervasively. Other problems that may have similar properties include properties of specific resource allocations, such as double locks or memory leaks. We are currently developing a general interprocedural program analysis framework targeting the Linux kernel in order to be apply the techniques used by DSAC to a wider range of bug types.

9 RELATED WORK

9.1 Detecting Concurrency Bugs

Many approaches [13, 21, 26, 35, 37, 44, 50, 55, 56] have been proposed to detect concurrency bugs in user-mode applications. Some of them [13, 26, 55] use dynamic analysis to collect and analyze runtime information to detect concurrency bugs. But the code coverage of dynamic analysis is limited by test cases. Others [21, 44, 50, 56] use static analysis to cover more code without running the tested programs. But static analysis often introduces false positives. Some approaches [14, 35, 37] combine static and dynamic analysis to achieve higher code coverage with fewer false positives. DSAC uses static analysis to cover the whole kernel, and validates the feasibility of code paths to reduce false positives.

To improve OS reliability, some approaches [20, 23, 25, 27, 53, 54, 57] detect some kinds of concurrency bugs like data races, but they do not detect SAC bugs. Several approaches [4, 15, 24, 46] can detect common kinds of OS kernel bugs, including SAC bugs. But they are not specific to SAC bugs, and most of them [15, 24, 46] are designed to collect statistics rather than report specific bugs to the user, making issues such as detection time and false positive rate less important. Besides, these approaches does not handle function pointers, so they may miss SAC bugs that involve function-pointer calls. For example, BlockLock [46] has an overall false positive rate of 20%, while that of DSAC is less than 10%, and BlockLock also misses many real bugs found by DSAC, especially the bugs involving function-pointer calls.

9.2 Checking API Rules

Checking API rules is a promising way of finding deep and semantic bugs in an OS kernel. Some approaches [6, 10, 40, 42] use known API rules to statically or dynamically detect API misuses. For example, with known paired kernel reference count management interfaces, RID [40] uses a summary-based inter-procedural analysis to detect reference counting bugs in the Linux kernel. To find implicit API rules, some approaches do specification mining by analyzing source code [33, 36, 38, 49, 59] or execution traces [8, 31, 58], and then use the mined API rules to detect violations. For example, PF-Miner [38] analyzes the source code of the Android kernel. It first identifies error handling paths from C source code, and then respectively collects function call sequences in normal execution paths and error handling paths. By using statistical methods to compare the collected function call sequences in the two kinds of code paths, PF-Miner extracts frequently used function pairs as API rules. Perracotta [58] analyzes execution traces for the Windows kernel. It partitions the original imperfect execution traces into some sub-traces, and identifies different objects using context-sensitive analysis. Then, Perracotta uses three heuristics, namely call-graph reachability, name similarity and properties combination, to find interesting API rules.

Most of these approaches focus on the temporal rules of common API usages, such as resource acquiring and releasing pairs [49, 58] and error handling patterns [8, 33, 38], but these approaches have not targeted SAC bugs.

9.3 Function-Pointer Analysis

In static analysis, analyzing function pointers is a classical and difficult problem, because the set of functions referenced by a function pointer is often hard to correctly identify without exact runtime information. Some alias-analysis approaches [22, 29, 30, 32, 41] have been proposed that can handle function pointers. They are classified as Andersen-style [3] or Steensgaard-style [51]. Andersen-style approaches view pointer assignments as subset constraints, and use constraints to propagate pointer information; Steensgaard-style approaches also use constraint-based analysis, but they use equality constraints instead of subset constraints. CLA [30] uses a field-based and flow-insensitive pointer analysis to analyze large-scale software in a short time. It is Andersen-style, and uses the data structure type and field name to maintain field sensitivity. DSA [32] is field-sensitive, context-sensitive and flow-insensitive pointer analysis algorithm. DSA is Steensgaard-style [51] with full heap cloning, and uses some practical optimizations to speed up alias analysis.

However, these approaches often identify incorrect referenced functions of function pointers in large and complex software. A main reason is that they do not consider the calling context of a function-pointer call, and just produce all the functions possibly referenced by the called function pointer. In this paper, based on the field-based analysis of CLA [30], we propose connection-based alias analysis to improve the accuracy of analyzing function-pointer calls.

9.4 Improving the Kernel Module Architecture

To prevent concurrency bugs in the OS kernel, several improved kernel module architectures have been proposed, typically for device drivers. The active driver architecture [2, 48] runs each driver in a separate kernel thread, and all communication between the driver and kernel is performed using message passing. This architecture can serialize the concurrent accesses to the driver and eliminate the possibility of concurrency bugs. The user-mode driver architecture [28, 34, 47] runs each driver in a separate user-mode process. This architecture protects the OS kernel against crashes caused by driver code. This architecture also allows the driver code to be implemented using a safer language such as Java [47], instead of C.

These approaches have two main limitations. Firstly, the source code of the kernel module must be manually rewritten. Secondly, the performance of the kernel module may degrade due to serialization in the active driver approaches and frequent context switches in the user-mode driver approaches.

10 CONCLUSION

In this paper, we have proposed DSAC, a practical static approach, to automatically and effectively detect SAC bugs in the Linux kernel. DSAC uses three key techniques: (1) a summary-based analysis to identify the code that may be executed in atomic context. (2) a connection-based alias analysis to identify the set of functions referenced by a function pointer. (3) a path-check method to filter out repeated reports and false bugs. We have used DSAC to check the kernel source code of Linux 4.17, and in total find 1068 real bugs. We have randomly selected 300 of the real bugs and sent them to kernel developers. 220 of these bugs have been confirmed, and 51 of our patches fixing 115 bugs have been applied.

DSAC can be still improved in several aspects. Firstly, the current implementation of DSAC can be improved to reduce false positives. For example, DSAC still fails to correctly validate the feasibility of the code path in complex cases. Thus, we need to improve our path-check method to accurately handle such cases. Secondly, at present, DSAC focuses on function pointers stored in data-structure fields, and cannot handle function pointers that are not referenced in this way. Thus, we need to improve our connection-based alias analysis to handle more kinds of function-pointer calls, which can help to find more SAC bugs. Thirdly, we have only evaluated DSAC on the Linux kernel in this paper. In fact, our previous DSAC approach [7] also found some real SAC bugs in the FreeBSD and NetBSD kernels. Thus, we will evaluate DSAC on these OS kernels. Finally, our connection-based alias analysis can help to build a full kernel call graph involving function-pointer calls. We are investigating whether the collected information is sufficient to accurately support detection of other kinds of kernel problems, such as double free and double lock bugs.

ACKNOWLEDGMENT

We would like to thank the Linux kernel developers and maintainers who gave helpful feedback on our bug reports and patches. This work was supported in part by the China Postdoctoral Science Foundation under Project 2019T120093. Shi-Min Hu is the corresponding author.

REFERENCES

[1] Allocation 2018. Linux kernel documentation for memory allocation. https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html.

[2] Sidney Amani, Peter Chubb, Alastair F Donaldson, Alexander Legg, Keng Chai Ong, Leonid Ryzhyk, and Yanjin Zhu. 2014. Automatic verification of active device drivers. *ACM SIGOPS Operating Systems Review* 48, 1 (2014), 106–118.

[3] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

[4] Zachary R Anderson, Eric A Brewer, Jeremy Condit, Robert Ennals, David Gay, Matthew Harren, George C Necula, and Feng Zhou. 2007. Beyond bug-finding: sound program analysis for Linux. In *Proceedings of the 11th International Workshop on Hot Topics in Operating Systems (HotOS)*. 1–6.

[5] Jia-Ju Bai, Julia Lawall, Wende Tan, and Shi-Min Hu. 2019. DCNS: automated detection of conservative non-sleep defects in the Linux kernel. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 287–299.

[6] Jia-Ju Bai, Hu-Qiu Liu, Yu-Ping Wang Wang, and Hu Shi-Min. 2014. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*. 407–414.

[7] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. 2018. DSAC: effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX ATC Conference (USENIX ATC)*. 587–600.

[8] Jia-Ju Bai, Yu-Ping Wang, Hu-Qiu Liu, and Shi-Min Hu. 2016. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology* 73 (2016), 122–133.

[9] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2011. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 8, 5 (2011), 670–684.

[10] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. 2006. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys)*. 73–85.

[11] BlockLock 2014. Website for "Faults in Linux: ten years later". http://faultlinux.lip6.fr/.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th International Conference on Operating Systems Design and Implementation (OSDI)*. 209–224.

[13] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A deployable sampling strategy for data race detection. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*. 810–821.

[14] Lee Chew and David Lie. 2010. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of 5th European Conference on Computer Systems (EuroSys)*. 307–320.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP)*. 73–88.

[16] Clang 2018. Clang compiler. http://clang.llvm.org/.

[17] CLOC 2018. CLOC: counting lines of code. https://github.com/AlDanial/cloc.

[18] Jonathan Corbet. 2008. Atomic context and kernel API design. https://lwn.net/Articles/274695/.

[19] Domenico Cotroneo, Roberto Natella, and Stefano Russo. 2009. Assessment and improvement of hang detection in the Linux operating system. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS)*. 288–294.

[20] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. 2015. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*. 166–177.

[21] Jyotirmoy Deshmukh, E Allen Emerson, and Sriram Sankaranarayanan. 2009. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*. 480–491.

[22] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 International Conference on Programming Language Design and Implementation (PLDI)*. 242–256.

[23] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*. 237–252.

[24] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th International Conference on Operating Systems Design and Implementation (OSDI)*. 1–16.

[25] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*. 151–162.

[26] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*. 215–228.

[27] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. 2014. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation (OSDI)*. 415–431.

[28] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. 2008. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 168–178.

[29] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*. 289–298.

[30] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 2001 International Conference on Programming Language Design and Implementation (PLDI)*. 254–263.

[31] Christopher LaRosa, Li Xiong, and Ken Mandelberg. 2008. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*. 880–885.

[32] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 International Conference on Programming Language Design and Implementation (PLDI)*. 278–289.

[33] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)*. 43–52.

[34] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. 2005. User-level device drivers: achieved performance. *Journal of Computer Science and Technology* 20, 5 (2005), 654–664.

[35] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2016. Effectively manifesting concurrency bugs in Android apps. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 209–216.

[36] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering (FSE)*. 306–315.

[37] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 677–691.

[38] Hu-Qiu Liu, Yu-Ping Wang, Jia-Ju Bai, and Shi-Min Hu. 2016. PF-Miner: a practical paired functions mining method for Android kernel in error paths. *Journal of Systems and Software* 121 (2016), 234–246.

[39] LLVM 2018. LLVM compiler infrastructure. https://llvm.org/.

[40] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 531–544.

[41] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2004. Precise call graphs for C programs with function pointers. *Automated Software Engineering* 11, 1 (2004), 7–26.

[42] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: the case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 361–377.

[43] MySQL 2018. MYSQL database. https://www.mysql.com/.

[44] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*. 308–319.

[45] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*. 247–260.

[46] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. 2014. Faults in Linux 2.6. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 4:1–4:40.

[47] Matthew J Renzelmann and Michael M Swift. 2009. Decaf: Moving device drivers to a modern language. In *USENIX Annual Technical Conference (USENIX ATC)*. 1–14.

[48] Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. 2010. The case for active device drivers. In *Proceedings of the 1st Aisa-Pacific Workshop on Systems (APSys)*. 25–30.

[49] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*. 1–12.

[50] Anirudh Santhiar and Aditya Kanade. 2017. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th International Conference on Programming Language Design and Implementation (PLDI)*. 292–305.

[51] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd International Symposium on Principles of Programming Languages (POPL)*. 32–41.

[52] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*. 207–222.
[53] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 11–20.
[54] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. 391–402.
[55] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*. 253–264.
[56] Amy Williams, William Thies, and Michael D Ernst. 2005. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*. 602–629.
[57] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. 2007. Model checking concurrent linux device drivers. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*. 501–504.
[58] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE)*. 282–291.
[59] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: sanitizing API usages through semantic cross-checking. In *USENIX Security Symposium*. 363–378.
[60] Yian Zhu, Yue Li, Jingling Xue, Tian Tan, Jialong Shi, Yang Shen, and Chunyan Ma. 2012. What is system hang and how to handle it. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*. 141–150.

Received October 2018; revised September 2019; accepted xxx